BAB III

PRINSIP ANALISA ALGORITMA

Dalam bab ini dibahas tentang fungsi pertumbuhan sebagai dasar untuk menentukan *running time. Running time* menunjukkan jumlah langkah yang digunakan algoritma untuk menyelesaikan permasalahan dengan input tertentu, serta beberapa prinsip analisa algoritma sebagai dasar untuk melihat kompleksitas waktu dari eksekusi suatu algoritma dari model *running time* yang dimiliki oleh suatu algoritma.

3.1. Dasar - Dasar Algoritma

Diberikan pendekatan dengan model matematika yang merupakan bagian dari penyelesaian masalah. Penyelesaian yang sempurna adalah sebuah metode yang dibutuhkan untuk menyelesaikan permasalahan umum dengan menggunakan model. Idealnya, Sebuah prosedur pada step - step barisan adalah peranan terpenting dari jawaban yang diinginkan.

Definisi 3.1

Sebuah Algoritma adalah prosedur terbatas untuk menyelesaikan sebuah permasalahan dengan langkah yang terbatas dalam waktu yang terbatas.

Istilah algoritma pertama kali ditulis oleh *al-Khowarizmi* matematikawan arab abad 19, dalam bukunya *Kitab al-jabr w'al muquabala*.

Diberikan sebuah algoritma untuk mencari pembagi bersama terbesar yang dikenal sebagai algoritma Euclids sebagai berikut :

Algoritma Euclids : Diberikan 2 (dua) bilangan bulat positif m dan n, untuk menemukan pembagi bersama terbesar yaitu bilangan bulat positif terbesar yang dapt membagi m dengan n.

E1. [Temukan sisa]. Bagi m dan n dan berikan r sebagai sisa pembagian (dalam hal ini $0 \leq r < n$).

E2. [Apakah sisanya nol]. Jika $r = 0$, algoritma berakhir, n sebagai jawaban.

E3. [Menukar tempat]. $m \leftarrow n$, $n \leftarrow r$, dan kembali kelangkah E1.

Ada beberapa sifat umum algoritma. Sifat - sifat tersebut digunakan untuk ketika sebuah algoritma dideskripsikan. Sifat - sifat tersebut adalah :

1. *Input*. Suatu algoritma dapat mempunyai nol atau banyak input, yaitu suatu kuantitas dimana diberikan nilai awal sebelum algoritma dimulai. Input

diperoleh dari himpunan objek yang telah ditetapkan. Dalam algoritma Euclids, terdapat 2 (dua) nilai input yang diberi nama m dan n, dimana keduanya diambil dari himpunan bilangan bulat positif.

2. *Output.* Dari setiap nilai input sebuah algoritma menghasilkan nilai output / keluaran dari himpunan yang dispesifikasikan. Nilai Output adalah penyelesaian dari permasalahan. Suatu algoritma dapat mempunyai satu atau banyak output. Algoritma Euclids mempunyai satu output, dinamakan n pada langkah E2, yang merupakan pembagi bersama terbesar dari dua input.

3. *Definiteness.* Langkah - langkah sebuah algoritma harus didefinisikan dengan tepat; tindakan penyelesaian harus ditetapkan dengan tepat dan jelas untuk setiap kasus. Dalam algoritma Euclids, pembacaan diperjelas dan mudah dimengerti apa arti pembagian m dengan n dan apa sisanya. Kriteria *definiteness* harus yakin bahwa nilai m dan n selalu bilangan bulat positif bilamana langkah E1 dieksekusi. Dengan hipotesa awal bernilai benar, dan setelah langkah E1, r adalah bilangan bulat non negatif dimana harus tidak nol jika didapatkan langkah E3, sehingga m dan n benar - benar bilangan bulat positif yang dibutuhkan.

4. *Finiteness.* Sebuah algoritma seharusnya menghasilkan output yang diinginkan setelah akhir dari langkah untuk sebarang himpunan input. Algoritma Euclids meyakinkan kondisi ini, karena setelah langkah E1 nilai r lebih kecil dari pada n, sehingga jika $r \neq 0$, nilai n berkurang untuk waktu selanjutnya bahwa langkah E1 ditemukan.

5. *Effectiveness.* Algoritma harus memungkinkan untuk setiap langkahnya dilakukan pada sebuah algoritma yang sebenarnya dan mempunyai jumlahan waktu yang terbatas. Artinya semua operasi yang dilakukan dalam algoritma adalah operasi dasar. Algoritma Euclids hanya digunakan operasi pembagian satu bilangan bulat positif dengan yang lain, pengujian if sebuah bilangan bulat nol dan letak nilai pada satu variabel sama dengan pada yang lain. Operasi ini adalah efektif, karena bilangan bulat dapat menggambarkan dalam cara terbatas dan terdapat minimal satu metode untuk membagi satu dengan yang lain. Tetapi untuk operasi yang sama mungkin tidak efektif jika nilai yang digunakan berbelit - belit untuk sebarang bilangan real yang ditentukan dengan ekspansi desimal tak terbatas. Contoh langkah yang tidak efektif : " Jika 2 bilangan bulat yang lebih besar dari n dimana ada sebuah solusi dengan persamaan sebagai $x^n + y^n = z^n$ dalam bilangan bulat positif x, y dan z, maka kembali kelangkah E4". Sehingga sebuah statement akan menjadi operasi yang tidak efektif sementara seseorang berhasil menunjukkan bahwa ada sebuah algoritma untuk menentukan apakah 2 atau tidak ada bilangan bulat yang lebih besar sebagaimana ditetapkan.

3.2. Fungsi Pertumbuhan

Diberikan sebuah program komputer dengan n bilangan bulat. Satu pertimbangan terpenting yang berkaitan dengan kegunaan program tersebut adalah berapa lama sebuah komputer menyelesaikan problem tersebut. Untuk

menganalisa kegunaan pada program, dibutuhkan pengertian bagaimana kecepatan fungsi pertumbuhan pada n pertumbuhan. Notasi yang sering digunakan untuk menganalisa fungsi pertumbuhan disebut notasi **big-O.** Hal ini untuk menunjukkan running time dari suatu program.

3.2.1. Notasi big-O

Fungsi pertumbuhan seringkali dideskripsikan dengan sebuah notasi khusus. Dengan mengikuti definisi dideskripsikan notasi tersebut.

Definisi 3.2

Diberikan f dan g suatu fungsi dari himpunan bilangan bilangan bulat atau himpunan bilangan real pada suatu himpunan bilangan real. Dikatakan f(x) adalah O(g(x)) jika terdapat sebuah konstanta C dan k sedemikian sehingga :

$$| f(x) | \leq C | g(x) |$$

dimana x > k. Dibaca f(x) adalah "**big-oh**" pada g(x).

Penjelasan :

Untuk menunjukkan f(x) adalah O(g(x)), hanya dibutuhkan untuk menemukan satu pasangan konstanta C dan k sedemikian sehingga $| f(x) | \leq C | g(x)|$ jika x>k. Pasangan C, k yang memenuhi definisi tidak pernah tunggal. Selanjutnya, jika satu pasangan ada, maka terdapat tak terbatas pasangan yang lain. Sebuah

cara sederhana untuk melihat hal tersebut adalah, jika C, k adalah satu pasangan, pasangan yang lain C′, k′ dengan $C < C'$ dan $k < k'$ juga memenuhi definisi, jika $| f(x) | \leq C |g(x)| \leq C' | g(x)|$ dimana $x > k' > k$.

Contoh 1:

Tunjukkan bahwa $f(x) = x^2 + 2x + 1$ adalah $O(x^2)$.

Penyelesaian :

Jika $0 \leq x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 = 4x^2$ dimana $x > 1$, mengikuti f(x) adalah $O(x^2)$. Untuk menerapkan definisi pada notasi big-O, ambil $C = 4$ dan $k=1$. Disini tidak perlu menggunakan nilai absolut jika semua fungsi pada persamaan ini adalah positif ketika x bernilai positif.

Pendekatan yang lain adalah jika $x > 2$, maka $2x \leq x^2$. Akibatnya, jika $x>2$, terlihat bahwa : $0 \leq x^2 + 2x + 1 \leq x^2 + x^2 + x^2 = 3x^2$. Dari definisi didapat $C = 3$ dan $k = 2$.

Pengamatan pada relasi f(x) adalah $O(x^2)$, x^2 dapat diganti dengan sebarang fungsi yang lain dengan nilai yang lebih besar dari x^2 , misalnya f(x) adalah $O(x^3)$, f(x) adalah $O(x^2+2x+7)$ dan seterusnya. Hal ini juga benar bahwa x^2 adalah $O(x^2+2x+1)$, jika $x^2 < x^2+2x+1$ dimana $x > 1$. ■

Dari contoh tersebut diperoleh dua fungsi, $f(x) = x^2 + 2x + 1$ dan $g(x) = x^2$, sedemikian sehingga f(x) adalah O(g(x)) dan g(x) adalah O(f(x)) - pernyataan terakhir dari pertidaksamaan $x^2 \leq x^2 + 2x + 1$, dimana untuk semua x bilangan real tidak negatif. Dikatakan kedua fungsi f(x) dan g(x) bahwa keduanya mengikuti relasi big-O dengan order yang sama.

Penjelasan :

Kenyataan bahwa f(x) adalah g(x) kadang - kadang ditulis f(x) = O(g(x)). Bagaimanapun, tanda sama dengan pada notasi tersebut tidak dapat direpresentasikan dengan sebuah persamaan sesungguhnya. Selanjutnya, notasi tersebut menggambarkan bahwa pertidaksamaan yang diperoleh menghubungkan nilai pada fungsi f dan g untuk bilangan yang cukup besar pada domain fungsi tersebut.

Notasi big-O digunakan pada bidang matematika, khususnya pada ilmu komputer untuk analisa algoritma. Matematikawan Jerman yang memperkenalkan pertama kali notasi big-O pada tahun 1892 pada sebuah buku tentang teori Bilangan. Notasi big-O kadang - kadang disebut juga dengan **Simbol Landau**.

Ketika f(x) adalah O(g(x)), dan h(x) adalah sebuah fungsi yang mempunyai nilai absolut lebih besar daripada g(x) untuk bilangan x yang cukup besar, mengikuti definisi diatas maka f(x) adalah O(h(x)). Dengan kata lain, fungsi g(x) pada relasi f(x) adalah O(g(x)) dapat direpresentasikan dengan sebuah fungsi

dengan nilai absolute yang lebih besar.

$| f(x) | \leq C | g(x)|$ jika $x > k$

dan jika $| h(x) | > | g(x) |$ untuk semua $x > k$, maka

$| f(x) | \leq C | h(x) |$ jika $x > k$.

Oleh karena itu f(x) adalah O(h(x)).

Ketika notasi big-O digunakan, fungsi g pada relasi f(x) adalah O(g(x)) dipilih dari nilai terkecil yang mungkin. Kadang - kadang dari sebuah himpunan fungsi tertentu, misalnya fungsi tersebut pada bentuk x^n , dimana n adalah bilangan bilangan bulat positif.

Contoh 2:

Tunjukkan bahwa $7 x^2$ adalah $O(x^3)$.

Penyelesaian :

Pertidaksamaan $7 x^2 < x^3$ dimana $x > 7$. Terlihat bahwa pembagian kedua sisi pertidaksamaan dengan x^2. Oleh karena itu, $7 x^2$ adalah $O(x^3)$, dengan mengambil $C = 1$ dan $k = 7$ pada definisi notasi big-O. ■

Contoh 3:

Pada contoh 2 ditunjukkan bahwa $7 x^2$ adalah $O(x^3)$. Apakah juga benar bahwa x^3 adalah $O(7 x^2)$?

Penyelesaian :

Untuk menentukan apakah x^3 adalah $O(7\,x^2)$, perlu ditentukan apakah terdapat konstanta C dan k sedemikian sehingga $x^3 \leq C\,(7\,x^2)$ dimana $x > k$. Pertidaksamaan tersebut ekuivalen dengan pertidaksamaan $x < 7\,C$, dimana dihasilkan dari pembagian kedua sisi dengan x^2. Tidak ada C jika x dibuat untuk sebarang bilangan yang lebih besar. Oleh karena itu x^3 bukan $O(7\,x^2)$. ■

Polinomial sering digunakan untuk mengestimasi fungsi pertumbuhan. Analisa pertumbuhan pada polinomial untuk setiap waktu , diperlukan hasil yang selalu dapat digunakan untuk mengestimasi pertumbuhan polinomial.

Teorema 3.1

Diberikan $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$, dimana $a_0 , a_1 , \ldots , a_{n-1} , a_n$ adalah bilangan real. Maka f(x) adalah $O(x^n)$.

Bukti :

Dengan menggunakan pertidaksamaan segitiga, jika $x > 1$, diperoleh :

$$
\begin{aligned}
|\, f(x) \,| &= |\, a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \,| \\
&\leq |a_n|\, x^n + |a_{n-1}|\, x^{n-1} + \ldots + |a_1|\, x + |a_0| \\
&= x^n \left(|a_n| + |a_{n-1}| / x + \ldots + |a_1| / x^{n-1} + |a_0| / x^n \right) \\
&\leq x^n \left(|a_n| + |a_{n-1}| + \ldots + |a_1| + |a_0| \right).
\end{aligned}
$$

Terlihat bahwa

$$| f(x) | \leq C x^n$$

dimana $C = |a_n| + |a_{n-1}| + \ldots + |a_1| + |a_0|$ untuk $x > 1$. Oleh karena itu, $f(x)$ adalah $O(x^n)$. ■

Contoh 4 :

Bagaimana notasi big-O digunakan untuk mengestimasi jumlahan pertama n bilangan bulat positif.

Penyelesaian :

Jika setiap bilangan bulat pada jumlahan pertama n bilangan bulat positif tidak melebihi n, mengikuti aturan diatas maka :

$1 + 2 + 3 + \ldots + n \leq n + n + \ldots + n = n^2$

Dari pertidaksamaan tersebut maka $1 + 2 + 3 + \ldots + n$ adalah $O(n^2)$, dengan mengambil $C = 1$ dan $k = 1$ pada definisi notasi big-O. Pada contoh ini domain fungsi big-O relasinya adalah himpunan bilangan bulat positif.

Contoh 5 :

Dapatkan estimasi big-O untuk fungsi faktorial dan logaritma pada fungsi faktorial, dimana fungsi faktorial $f(n) = n !$ didefinisikan dengan :

$$n ! = 1 . 2 . 3 . \ldots . n$$

dimana n adalah bilangan bulat positif, dan 0 ! = 1, Sebagai contoh :

1 ! = 1 , 2 ! = 1 . 2 = 2 , 3 ! = 1 . 2 . 3 = 6 dan seterusnya

Penyelesaian :

Sebuah estimasi big-O pada n ! dapat diperoleh dengan catatan bahwa setiap hubungan dalam perkalian tidak melebihi n. Sehingga :

$$n ! = 1 . 2 . 3 . . . . n$$

$$\leq n . n . n . . . . . n$$

$$= n^n$$

Pertidaksamaan tersebut menunjukkan bahwa n ! adalah $O(n^n)$. Dengan mengambil logaritma dari kedua sisi pertidaksamaan terbukti untuk n !, diperoleh:

$$\log n ! \leq \log n^n = n \log n$$

Secara tidak langsung bahwa log n ! adalah $O(n^n)$.

3.2.2. Kombinasi Fungsi Pertumbuhan

Banyak algoritma dibuat pada dua atau lebih sub prosedur. Jumlah langkah yang digunakan oleh komputer untuk menyelesaikan sebuah permasalahan dengan input / masukkan tertentu pada sebuah algoritma adalah jumlahan setiap langkah yang digunakan oleh sub prosedur tersebut. Untuk mendapatkan estimasi big-O pada jumlah langkah yang dibutuhkan, sangat perlu

untuk menemukan estimasi big-O pada jumlah langkah yang digunakan pada setiap sub prosedur dan selanjutnya estimasi tersebut dikombinasikan.

Estimasi big-O pada kombinasi fungsi dapat ditetapkan jika ketelitian yang diambil berbeda estimasi big-O nya untuk digabungkan. Khususnya, sangat perlu untuk mengestimasi pertumbuhan pada jumlahan dan perkalian dua fungsi. Misalkan $f_1(x)$ adalah $O(g_1(x))$ dan $f_2(x)$ adalah $O(g_2(x))$, dari definisi notasi big-O, disana terdapat konstanta C_1 , C_2 , k_1 dan k_2 sedemikian sehingga :

$$| f_1(x) | \leq C_1 | g_1(x) |$$

ketika $x > k_1$ dan

$$| f_2(x) | \leq C_2 | g_2(x) |$$

ketika $x > k_2$. Untuk mengestimasi jumlahan pada $f_1(x)$ dan $f_2(x)$, perhatikan bahwa

$| (f_1 + f_2) (x) | = | f_1(x) + f_2(x) |$

$\leq | f_1(x) | + | f_2(x) |$, dengan menggunakan pertidaksamaan segitiga $| a + b | \leq | a | + | b |$.

Ketika x lebih besar dari k_1 dan k_2, dengan mengikuti pertidaksamaan untuk $| f_1(x) |$ dan $| f_2(x) |$ maka :

$$| f_1(x) | + | f_2(x) | < C_1 | g_1(x) | + C_2 | g_2(x) |$$

$$\leq C_1 | g(x) | + C_2 | g(x) |$$

$$= (C_1 + C_2) | g(x) |$$

$$= C | g(x) |$$

dimana $C = C_1 + C_2$ dan $g(x) = \max (| g_1(x) | , | g_2(x) |)$.

Disini, max (a , b) menunjukkan maksimum atau lebih besar pada a dan b.

Pertidaksamaan tersebut menunjukkan bahwa $| (f_1 + f_2) (x) | \leq C | g(x) |$ dimana $x > k$, dimana $k = \max (k_1 , k_2)$.

Hasil tersebut digunakan sebagai teorema sebagai berikut :

Teorema 3.2

Diberikan $f_1(x)$ adalah $O(g_1(x))$ dan $f_2(x)$ adalah $O(g_2(x))$. Maka $(f_1(x) + f_2(x))$ (x) adalah $O(\max(g_1(x) , g_2(x)))$. Selanjutnya disebut juga dengan **Maximum rule.**

Seringkali dipunyai estimasi big-O untuk f_1 dan f_2 pada hubungan tersebut mempunyai fungsi yang sama yaitu g. Pada situasi ini, teorema 3.2 dapat digunakan untuk menunjukkan bahwa $(f_1 + f_2) (x)$ adalah juga $O(g(x))$, jika $\max (g(x) , g(x)) = g(x)$. Hasil tersebut ditetapkan sebagai akibat.

Akibat teorema 3.2

Diberikan $f_1(x)$ dan $f_2(x)$ keduanya adalah $O(g(x))$. Maka $(f_1 + f_2) (x)$ adalah $O(g(x))$.

Dengan cara yang sama estimasi big-O dapat diperoleh dengan perkalian fungsi f_1 dan f_2 . Ketika x lebih besar daripada max (k_1 , k_2), maka :

$$| (f_1 f_2) (x) | = | f_1(x) | | f_2(x) |$$

$$\leq C_1 | g_1(x) | \; C_2 | g_2(x) |$$

$$\leq C_1 \; C_2 | (g_1 \; g_2) (x) |$$

$$\leq C | (g_1 \; g_2) (x) |$$

dimana $C = C_1 \; C_2$. Dari pertidaksamaan tersebut, dengan mengikuti definisi maka $f_1(x)f_2(x)$ adalah $O(g_1 \; g_2)$, jika disana terdapat konstanta C dan k, yaitu $C = C_1 \; C_2$ dan $k=\max(k_1 , k_2)$, karena $| (f_1 f_2) (x) | \leq C | (g_1 \; g_2) (x) |$ dimana $x > k$. Hasil tersebut ditetapkan sebagai teorema berikut :

Teorema 3.3

Diberikan $f_1(x)$ adalah $O(g_1(x))$ dan $f_2(x)$ adalah $O(g_2(x))$. Maka $(f_1 f_2) (x)$ adalah $O(g_1(x)g_2(x))$.

Tujuan menggunakan notasi big-O pada fungsi estimasi adalah untuk memilih sebuah fungsi g(x) pada pertumbuhan relatif terendah dengan f(x) adalah O(g(x)).

Contoh 6 :

Dapatkan estimasi big-O untuk $f(n) = 3 n \log(n !) + (n^2 + 3) \log n$, dimana n adalah bilangan bilangan bulat positif.

Penyelesaian :

Pertama, Perkalian 3 n log(n !) akan diestimasi. Dari contoh 5 diketahui bahwa log(n!) adalah O(n log n). Menggunakan estimasi tersebut dan 3n adalah O(n). Teorema 3.3 memberikan estimasi bahwa 3 n log(n !) adalah $O(n^2 \log n)$.

Selanjutnya, perkalian $(n^2 + 3) \log n$ akan diestimasi. Untuk $(n^2+3)<2n^2$, maka $n > 2$, sehingga $(n^2 + 3)$ adalah $O (n^2)$. Maka, dari teorema 3.3 , $(n^2 + 3) \log n$ adalah $O(n^2 \log n)$. Menggunakan teorema 3.2 untuk menggabungkan dua estimasi big-O untuk perkalian terlihat bahwa $f(n) = 3 n \log(n !) + (n^2 + 3) \log n$ adalah $O(n^2 \log n)$.

Contoh 7 :

Dapatkan estimasi big-O untuk $f(x) = (x + 1) \log(x^2 + 1) + 3 x^2$

Penyelesaian :

Pertama, estimasi big-O untuk $(x + 1) \log(x^2 + 1)$ akan diperoleh. Catatan (x + 1) adalah O(x). selanjutnya, $x^2 + 1 \leq 2 x^2$ ketika $x > 1$. Sehingga, $\log(x^2 + 1) \leq \log(2 x^2) = \log 2 + \log x^2 = \log 2 + 2 \log x \leq 3 \log x$, jika $x > 2$. Ini terlihat bahwa $\log(x^2 + 1)$ adalah $O(\log x)$.

Dari teorema 3.3 bahwa $(x + 1) \log(x^2 + 1)$ adalah $O(x \log x)$. Untuk $3 x^2$ adalah $O(x^2)$, teorema 3.2 menunjukkan bahwa f(x) adalah $\max(O(x \log x , x^2))$. Padahal $x \log x \leq x^2$, untuk $x > 1$, sehingga f(x) adalah $O(x^2)$.

3.3. Analisa Algoritma

Jika ditemukan beberapa algoritma yang berbeda untuk menyelesaikan permasalahan yang sama, maka harus dipilih salah satu yang sesuai dengan permintaannya. Alat yang paling utama untuk tujuan tersebut adalah *Analisa Algoritma*. Setelah menentukan efisiensi bermacam - macam algoritma yang dapat membuat keputusan terbaik untuk digunakan, tetapi tidak ada formula khusus untuk menganalisa efisiensi algoritma. Sebagian besar berdasarkan pendapat, intuisi dan pengalaman. Meskipun demikian, ada beberapa teknik dasar yang sering digunakan, seperti pengetahuan tentang memperlakukan struktur kontrol dan persamaan rekurensi.

3.3.1. Analisa Struktur Kontrol

Analisa algoritma biasanya meneruskan dari bagian dalam. Pertama, menetapkan waktu yang diperlukan oleh instruksi tunggal (seringkali dibatasi dengan konstanta), kemudian menggabungkan waktu tersebut berdasarkan struktur kontrol dan menggabungkan instruksi - instruksi pada program.

3.3.1.1. Barisan *(Sequencing)*.

Diberikan $\mathbf{P}_1$ dan $\mathbf{P}_2$ adalah dua penggalan pada sebuah algoritma. Mungkin merupakan instruksi tunggal atau subalgoritma yang rumit susunannya. Diberikan $\mathbf{t}_1$ dan $\mathbf{t}_2$ adalah waktu yang yang berturut - turut dimiliki oleh $\mathbf{P}_1$ dan

P_2. Waktu tersebut akan tergantung pada bermacam - macam parameter, seperti ukuran kejadian. **Aturan Sequencing** dikatakan membutuhkan waktu untuk menghitung "*P_1 dan P_2*", maka pertama P_1 dan selanjutnya P_2 , secara sederhana $t_1 + t_2$. Dengan menggunakan aturan maksimum *(maximum rule)* , waktu tersebut adalah $O(\max(t_1 , t_2))$.

3.3.1.2. Loop "*For*"

Loop **For** adalah loop termudah untuk dianalisa, dibandingkan dengan loop - loop yang lain.

for i ← 1 **to** m **do** P(i)

Misalkan loop tersebut bagian dari sebuah algoritma yang besar, bekerja dengan ukuran kejadian n. Kasus termudah adalah ketika waktu yang dibutuhkan oleh P(i) tidak tergantung pada i, meskipun dapat tergantung pada ukuran kejadian atau secara umum pada kejadiannya sendiri. Diberikan t yang merupakan waktu yang dibutuhkan untuk menghitung P(i). Pada kasus ini, jelas bahwa pada loop P(i) digunakan m kali, setiap waktu berharga t, dan total waktu yang dibutuhkan oleh loop adalah 1 = m t. Meskipun biasanya pendekatan ini kurang memadai, disini mempunyai kesukaran yang tersembunyi, karena tidak dapat diuraikan waktu yang dibutuhkan untuk **loop kontrol.** Loop **For** adalah yang terpendek dibandingkan dengan loop **While.**

```
i ← 1
while i ≤ m do
        P(i)
        i ← i + 1
```

Pada banyak keadaan, jumlah yang wajar suatu harga unit test $i \leq m$ adalah instruksi $i \leftarrow 1$ dan $i \leftarrow i + 1$, dan operasi berantai (**go to**) sudah termasuk dalam loop **while**. Diberikan c adalah batas atas waktu yang dibutuhkan oleh setiap operasi. Waktu 1 yang digunakan oleh loop dengan batas diatas adalah

$1 \leq$		c	untuk $i \leftarrow 1$
	$+$	$(m + 1)\, c$	untuk test $i \leq m$
	$+$	$m\, t$	untuk eksekusi pada P(i)
	$+$	$m\, c$	untuk eksekusi pada $i \leftarrow i + 1$
	$+$	$m\, c$	untuk operasi berantai
$\leq$		$(t + 3\, c)\, m + 2\, c.$	

Selain itu waktu tersebut dengan jelas mempunyai batas bawah mt. Jika c diabaikan untuk dibandingkan dengan t, perkiraan sebelumnya bahwa 1 kira - kira sama dengan mt adalah bernilai benar, kecuali pada satu kasus kritis : $1 \approx mt$ adalah kesalahan kompleks ketika $m = 0$ (lebih buruk lagi jika m adalah negatif).

Analisa loop **for** akan lebih menarik ketika waktu yang dibutuhkan t(i) pada P(i) berubah - ubah berdasar fungsi pada i. Secara umum, waktu yang

dibutuhkan P(i) tidak hanya tergantung pada i tetapi juga pada ukuran kejadian n. Jika diabaikan waktu dalam loop kontrol, dimana biasanya cukup $m \geq 1$, maka loop **for** sama dengan

for i ← 1 **to** m **do** P(i)

waktu yang diberikan tidak sama dengan perkalian tetapi mirip dengan penjumlahan , penjumlahan tersebut adalah :

$$\sum_{i=1}^{m} t(i)$$

3.3.1.3. Prosedur Rekursif

Analisa algoritma rekursif biasanya langsung, langkah demi langkah sampai akhir.

function Fibrec(n)

if n < 2 **then return** n

else return Fibrec(n-1)+Fibrec(n-2)

Diberikan T(n) adalah waktu yang diperlukan untuk memanggil Fibrec(n). Jika n < 2, algoritma dengan sederhana kembali ke n, dimana memberikan waktu suatu konstanta a. Sebaliknya, sebagian besar pekerjaan yang dipakai adalah memanggil dua rekursif, dimana waktunya berturut - turut T(n-1) dan T(n-2). Selain itu, satu penambahan melibatkan f_{n-1} dan f_{n-2} harus dilakukan, dan juga

kontrol pada rekursi dan test "**if** n < 2". Diberikan h(n) untuk menunjukkan kerja yang dilibatkan pada penambahan tersebut dan kontrol bahwa waktu yang dibutuhkan untuk memanggil Fibrec(n) diabaikan waktu yang dipakai untuk memanggil di dalam dua rekursif . Dari definisi T(n) dan h(n) diperoleh rekurensi:

$$T(n) = \begin{cases} a & \text{jika } n = 0 \text{ atau } n = 1 \\ T(n-1) + T(n-2) + h(n) & \text{yang lain} \end{cases}$$

3.3.1.4. Loop while dan repeat

Loop **While** dan **repeat** biasanya lebih sulit untuk dianalisa daripada loop **for** karena tidak ada langkah pasti untuk mengetahui berapa banyak waktu akan dipunyai untuk dilalui loop. Teknik standar untuk menganalisa loop tersebut adalah dengan menemukan fungsi pada variabel yang terlibat mana yang mengurangi nilai setiap waktu. Untuk mengakhiri bahwa loop akan berakhir, dicukupkan untuk diperlihatkan suatu nilai harus bilangan bilangan bulat positif.

Akan diperlihatkan *binary search*. Yang selanjutnya akan dianalisa dengan loop **while.** Tujuan dari *binary search* adalah menemukan sebuah elemen x pada array T[1..n] mempunyai order tetap. Asumsi untuk mempermudah bahwa x adalah dijamin terlihat minimal sekali pada T. Kebutuhan untuk menemukan sebuah bilangan bulat i dimana $1 \leq i \leq n$ dan T[i] = x. Ide dasar *binary search*

adalah membandingkan x dengan elemen y pada pertengahan T. Pencarian adalah berakhir jika x = y , ini dapat dibatasi untuk setengah batas atas pada array jika x > y, sebaliknya, ini tidak cukup untuk pencarian setengah batas bawah.

```
function Binary_search(T[1..n],x)
i ← 1 ; j ← n
while i < j do
    {T[i] ≤ x ≤ T[j]}
    k ← ( i + j ) / 2
        case x < T[k] : j ← k - 1
             x = T[k] : i, j ← k {return k}
             x > T[k] : i ← k + 1
    return i
```

Mengingat kembali analisa running time pada loop **while**, harus menemukan fungsi pada variabel yang terlibat mana yang mengurangi nilai setiap waktu pada loop. Pada kasus ini, umumnya menganggap j-i+1, dimana akan disebut d. Maka d menggambarkan bilangan pada elemen T yang masih dibawah pertimbangan. Awalnya, d = n. Loop berakhir ketika i ≥ j, dimana ekuivalen pada d ≤ 1. Pada kenyataannya, d tidak pernah lebih kecil daripada 1. Setiap waktu dalam perputaran loop, akan terjadi tiga kemungkinan : tiap j adalah himpunan pada k-1, i adalah himpunan pada k+1 atau keduanya i dan j adalah himpunan pada k.

Diberikan d dan $\hat{d}$ berturut - turut sebagai pemberhentian untuk nilai pada j-i+1 sebelum dan sesudah iterasi dibawah pertimbangan.

Gunakan i , j , $\hat{i}$ dan $\hat{\supset}$ adalah identik. Jika x < T[k], instruksi "j ← k-1" dieksekusi

dan $\hat{i}$ = i dan $\hat{\supset}$ = [(i+j)/2]-1 . Oleh karena itu,

$$\hat{d} = \hat{\supset} - \hat{i} + 1 = (i + j)/2 - i < (j - i + 1)/2 = d/2$$

Identik, jika x > T[k], instruksi "i ← k + 1" dieksekusi dan sehingga

$$\hat{i} = [(i + j)/2] + 1 \quad \text{dan} \quad \hat{\supset} = j$$

Oleh karena itu,

$$\hat{d} = \hat{\supset} - \hat{i} + 1 = j - (i + j)/2 \leq j - (i + j - 1)/2 = d/2$$

Akhirnya, jika x = T[k], maka i dan j adalah himpunan yang mempunyai nilai sama sehingga

$\hat{d}$ = 1, tetapi d < 2 jika sebaliknya loop tidak masuk kembali.

Disimpulkan $\hat{d} \leq d/2$ untuk sebarang kasus, dimana rata - rata nilai d lebih kecil dari setengah waktu sekitar loop. Jika berakhir ketika $d \leq 1$, maka proses akhirnya harus berhenti.

Untuk menentukan sebuah batas atas *running time binary search,* diberikan d_l merupakan nilai j - i + 1 pada akhir putaran ke-l loop untuk $l \geq 1$ dan diberikan $d_0 = n$. Jika d_{l-1} adalah nilai pada j - i + 1 sebelum dimulai iterasi ke-l, harus ditunjukkan bahwa $d_l \leq d_{l-1}/2$ untuk semua $l \geq 1$. Dengan menggunakan induksi matematik bahwa $d_l \leq n/2^l$. Tetapi loop berakhir ketika $d \leq 1$, dimana terjadi terakhir ketika $l = [\lg n]$. Kesimpulannya sebagian besar loop mengikuti waktu $\lceil \lg n \rceil$ Jika setiap putaran sekitar loop memberikan waktu konstan, *binary search* memberikan waktu O(log n).

3.3.2. Aturan Umum Analisa Algoritma

Secara umum, running time pada sebuah statemen atau kelompok statemen mungkin terparameterisasi dengan ukuran input dan atau dengan banyak variabel. Parameter yang diperbolehkan untuk running time pada keseluruhan program adalah n ukuran input. Aturan umum untuk analisa algoritma adalah sebagai berikut :

1. running time setiap **assigment** (tugas), baca (**read**) dan statemen **Write** besarnya dapat diambil O(1).

2. running time pada barisan statemen ditentukan dengan aturan jumlahan yaitu bahwa running time pada barisan adalah tidak melebihi dari sebuah faktor konstan yang merupakan running time terbesar pada beberapa statement barisan.
3. running time pada statemen **if** adalah harga pada kondisi statemen eksekusi. Waktu menghitung kondisi secara normal adalah O(1). Waktu untuk **if then else** adalah waktu menghitung kondisi ditambah waktu terbesar yang dibutuhkan untuk statemen eksekusi jika kondisinya false (salah).
4. Waktu eksekusi adalah jumlah semua waktu sekitar loop, waktu eksekusi sekumpulan statemen dan waktu mengevaluasi kondisi untuk penghentian (biasanya yang terakhir adalah O(1)).

Untuk selanjutnya f(x) disebut pula dengan T(n) atau running time.

Contoh Analisa Algoritma Sorting, diberikan prosedur program sebagai berikut:

```
procedure select(T[1..n])
    for i ← 1 to n - 1 do
        minj ← i ; minx ← T[i]
        for j ← i + 1 to n do
            if T[j] < minx then minj ← j
                                minx ← T[j]
                                T[minj] ← T[i]
                                T[i] ← minx
```

Meskipun waktu yang diperlukan oleh setiap perputaran didalam loop bukan konstanta, waktu tersebut adalah dibatasi oleh suatu konstanta c. Untuk setiap nilai pada i, instruksi didalam loop yang dieksekusi adalah n - (i + 1) + 1 = n-i kali, dan selanjutnya waktu yang diberikan oleh loop terdalam adalah $t(i) \leq (n - i) c$. Waktu yang yang diberikan oleh putaran ke-i pada loop terluar dibatasi oleh b + t(i) untuk sebuah konstanta b memberikan jumlahan pada oprerasi - operasi dasar sebelum dan sesudah loop terdalam dan loop kontrol untuk loop terluar. Selanjutnya, total waktu yang diperlukan oleh algoritma diatas adalah

$$\sum_{i=1}^{n-1} b + (n - i) c = \sum_{i=1}^{n-1} (b + c n) - c \sum_{i=1}^{n-1} i$$

$$= (n - 1) (b + c n) - c n (n - 1) / 2$$

$$= \frac{1}{2} c n^2 + \left(b - \frac{1}{2} c \right) n - b$$

adalah $O(n^2)$.

3.3.3. Kompleksitas Algoritma

Salah satu alat yang digunakan untuk mengetahui efisiensi sebuah algoritma adalah waktu yang digunakan oleh komputer untuk menyelesaikan sebuah permasalahan dengan menggunakan algoritma tersebut, ketika sebuah nilai input ukurannya telah dispesifikasikan.

Pertanyaan tentang hal tersebut diatas termasuk dalam **Kompleksitas perhitungan** sebuah algoritma. Sebuah analisa waktu yang diinginkan untuk menyelesaikan sebuah permasalahan pada ukuran tertentu termasuk dalam **Kompleksitas waktu** sebuah algoritma. Sebuah analisa memory yang diinginkan termasuk dalam **Kompleksitas ruang** pada algoritma. Pertimbangan kompleksitas waktu dan ruang pada sebuah algoritma adalah penting ketika sebuah algoritma akan diimplementasikan. Jelas sekali, penting untuk diketahui ketika sebuah algoritma akan menghasilkan jawaban dalam mikrosecond, menit atau jutaan tahun. Demikian juga, memory yang diinginkan harus tersedia untuk menyelesaikan suatu permasalahan, sehingga kompleksitas ruang harus dapat digunakan.

Kompleksitas waktu pada sebuah algoritma dapat diekspresikan pada jumlah operasi yang digunakan oleh algoritma ketika input mempunyai ukuran tertentu. Operasi - operasi yang digunakan untuk mengukur kompleksitas waktu dapat berupa perbandingan bilangan bulat, penambahan bilangan bulat, perkalian bilangan bulat, pembagian bilangan bulat ataupun sebarang operasi dasar lain.

Contoh :

Diberikan algoritma untuk mencari elemen maksimum sebagai berikut :

procedure *max*(a_1 , a_2 , . . . , a_n : *bilangan bulat*)

max := a_1

for *i* :=*2* **to** *n*

if *max* < a_i **then** *max* := a_i

Penyelesaian :

Bilangan yang dibandingkan akan digunakan untuk mengukur kompleksitas waktu algoritma, karena perbandingan adalah operasi dasar yang digunakan.

Untuk menemukan elemen maksimum pada sebuah himpunan dengan n elemen, daftar mempunyai sebarang order, kadang - kadang elemen maksimum ada pada himpunan pertama sama dengan inisial pada daftar. Maka, setelah dibandingkan akan diperoleh bahwa akhir daftar bukan daerah jangkauan, kadang - kadang maksimum dan suku kedua dibandingkan, selanjutnya nilai maksimum pada elemen kedua jika ini yang terbesar. Prosedur ini bersambung, menggunakan dua penambahan perbandingan untuk setiap term pada daftar. Jika dua perbandingan digunakan untuk setiap selesai langkah kedua pada n elemen dan satu lagi perbandingan digunakan untuk keluar dari loop ketika i = n + 1, tepat 2 (n - 1) + 1 = 2n - 1 perbandingan digunakan pada aplikasi algoritma tersebut. Sehingga, algoritma untuk menemukan elemen maksimum pada

himpunan dengan n elemen mempunyai kompleksitas waktu O(n), yang merupakan ukuran pada term - term perbandingan bilangan yang digunakan.

Tabel 1 menunjukkan beberapa terminologi yang digunakan untuk mendeskripsikan kompleksitas waktu algoritma. Sebuah algoritma dikatakan mempunyai **Kompleksitas ekponensial** jika mempunyai waktu kompleksitas $O(b^n)$, dimana $b > 1$. Dengan cara yang sama, sebuah algoritma dengan waktu kompleksitas $O(n^b)$ dikatakan mempunyai **kompleksitas polynomial.**

Tabel 1

Terminologi yang digunakan untuk Kompleksitas Algoritma

Kompleksitas	Terminologi
O(1)	Kompleksitas Konstanta
O(log n)	Kompleksitas Logaritma
O(n)	Kompleksitas Linier
O(n log n)	Kompleksitas n log n
$O(n^b)$	Kompleksitas Polynomial
$O(b^n)$, dimana $b > 1$	Kompleksitas Eksponensial
O(n!)	Kompleksitas Faktorial

Diambil dari buku "Discrete Mathematics and its Applications", Kenneth H. Rossen, edisi tiga halaman 108

Sebuah estimasi big-O pada kompleksitas waktu algoritma menyatakan bagaimana waktu yang diinginkan untuk menyelesaikan permasalahan berubah sesuai dengan pertambahan ukuran input. Dalam prakteknya, estimasi terbaik yang digunakan (dengan fungsi referensi terkecil) dapat ditunjukkan. Estimasi big-O pada kompleksitas waktu tidak dapat diterjemahkan secara langsung kedalam penjumlahan aktual waktu yang digunakan komputer. Satu alasan adalah bahwa estimasi big-O f(n) adalah O(g(n)), dimana f(n) adalah kompleksitas waktu sebuah algoritma dan g(n) adalah fungsi referensi, sehingga $f(n) \leq C\, g(n)$ ketika $n > k$, dimana C dan k adalah konstanta. Sehingga tanpa diketahui konstanta C dan k pada pertidaksamaan, estimasi ini tidak dapat digunakan untuk menentukan batas atas jumlah operasi yang digunakan. Selanjutnya, waktu yang diinginkan untuk suatu operasi tergantung pada tipe operasi dan komputer yang digunakan.